

**Subject:**

Parallel and Distributed Computing

**Teacher:**

Sir Nadeem Kafi

**Group members:**

Mubin 20K-0211

Faizan 20K-0171

Fahad 20K-0441

**Project:**

Sorting algorithms (Bubble and Merge Sorts) implementation using OPENMP and MPI

**System Configuration**

Processor Model: Intel i5 3570

Processor Cores/Threads: 4/4

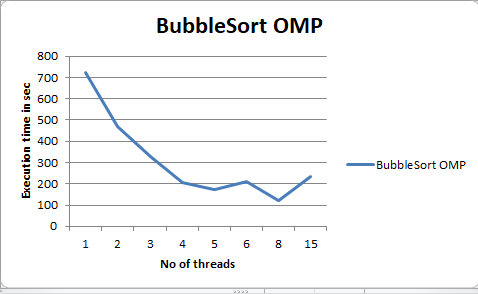
Processor Frequency: @3400 MHz

System Memory: 10 GB

Memory Frequency: 1333 MHz

**BUBBLE SORT:**

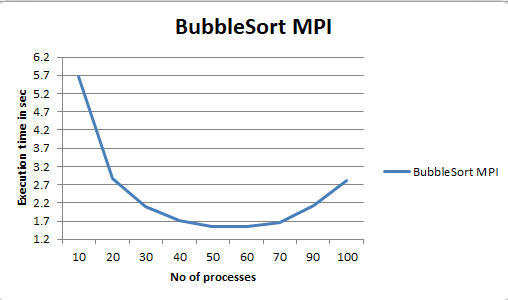
1. **Implemented in OPENMP:**



**Implementation**: Bubble sort algorithm was converted in openmp by parallelizing for loop of the algorithm using “pragma openmp parallel for ” where the shared variable where array itself, size and I and j was a private variable between threads.

**Result**: We can see to sort an array size of 250k the minimum time taken was more than 120 second at 8 threads and increasing the thread gradually increases the execution time because of overhead and thread communication.

1. **Code made in MPI:**



**Implementation**: A master process creates an array of 250k and distributes the chunk size to the processes using broadcast. Then the array is scattered to each process in chunks of defined size. The processes sort their chunk and send the partial result to the master process which then sorts these chunks again to form the final sorted data set.

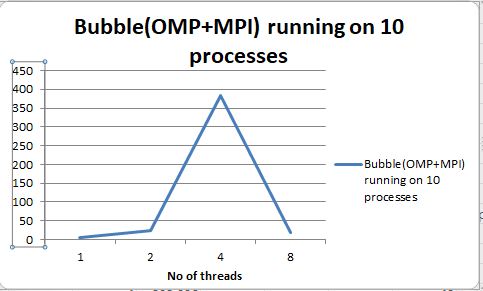
Log file is updated with the number of processes and execution time and the sorted array is also stored in a file to review the output.

**Result**: The mpi code created by our group shows a massive improvement as the minimum time to sort the same size array drops down to 1.5 sec using 50 processes and previously it was 210 sec. Maximum time it takes to sort an array using bubble sort(mpi) is 5.7 second which is even less than the best time of sorting through openmp.

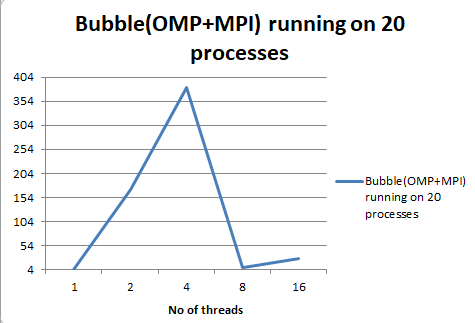
1. **Code made using OPENMP and MPI methods combined, i.e using varying threads with varying processes.**

**Implementation**: The MPI code was modified and merged with the initial OPENMP code to ensure proper working of both. The number of threads are passed as command-line arguments instead of console input. The MPI code first scatters the data as before, then the OPENMP threads further parallelise the workflow within each process.

**Results**: In the first graph we can see that we are using varying thread numbers with the process size of 10. The optimum combination shown is to use 1 thread per process which takes the lowest execution time i.e is 5 sec and less.

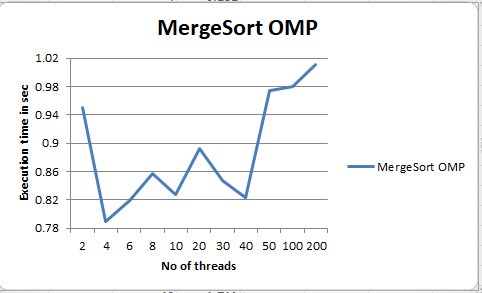


The next graph shows the varying threads with 20 processes and minimum time taken was less than 4 seconds for 20 threads per 20 processes.



**MERGE SORT:**

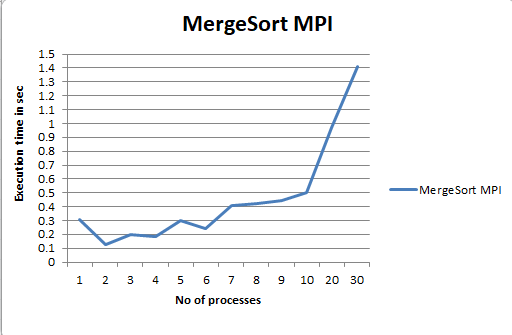
1. **Implementation in OPENMP:**



**Implementation:** Merge sort is a recursive divide and conquer algorithm which is significantly faster than bubble sort. The openmp implementation further improves on this by parallelising each recursive call using tasks. Then the merge function is executed to compile the final output array and follows a barrier before it, so each thread finishes before merging the array.

**Result**: For sorting 1 million numbers the minimum time merge sort takes using openmp is 0.79 sec with 4 threads and maximum time it takes was less than second on 200 threads but increasing the thread numbers after 40 exponentially increases the execution time. Therefore keeping thread below 40 would be best to sort 1 million numbers.

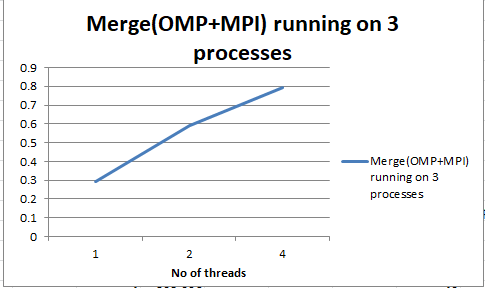
1. **Code made using MPI:**

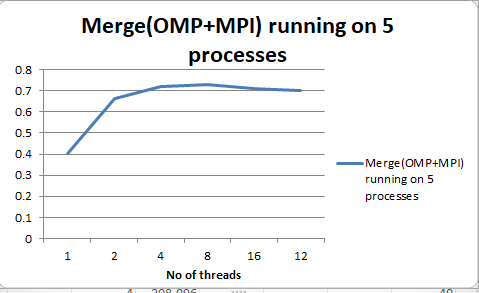


**Implementation**: The idea is similar to bubble sort as we scatter the data elements of specific chunk size to all processes and call the merge sort function which divides the array and sorts it. Each process undergoes this process and then the master process again calls the function to sort the individual chunks.

**Result**: The sorting time dropped to less than 0.1 second to sort 1 million numbers which was previously taking almost 0.79 seconds. Increasing the number of processes after 10 gradually increases the overhead of communication and increases the execution time.

1. **Code made using OPENMP and MPI methods combined, i.e using varying threads with varying processes.**





**Implementation**: The MPI code is further optimized using OPENMP for each recursive call within each process after the processes receive their data set from the master process. This parallelizes each call and runs it on the defined number of threads passed as command-line arguments for every process.

**Result**: First image no of process was fixed to 3 with varying thread numbers and the minimum time was 0.29 seconds and in the second image no of processes was fixed to 5 with varying thread resulting in minimum time to 0.5 seconds.

**CONCLUSION**

Data sizes: Bubble Sort 250k, Merge Sort 1M

**Minimum Execution time:**

| **Algorithm** | **OPENMP** | **MPI** | **OPENMP+MPI** |
| --- | --- | --- | --- |
| **Bubble** | 123.68 s  (6 threads) | 1.548 s  (60 procs) | 4.62 s  (20 procs + 1 thread/process) |
| **Merge** | 0.79 s  (4 threads) | 0.18 s  (5 procs) | 0.295 s  (3 procs + 1 thread/process) |

**Maximum Execution time:**

| **Algorithm** | **OPENMP** | **MPI** | **OPENMP+MPI** |
| --- | --- | --- | --- |
| **Bubble** | 722.46 s  (2 threads) | 5.67 s  (20 procs) | 383.25 s  (20 procs + 4 thread/process) |
| **Merge** | 1.01 s  (200 threads) | 1.48 s  (30 procs) | 1.87 s  (10 procs + 2 thread/process) |

By the results above we can conclude that the mpi method would yield the best results and the codes made by our group (mpi and openmp+mpi combined) shows drastic improvement than the openmp results. In theory, the combination of MPI with OPENMP should reduce the overall time consumption, however in our case it has an adverse effect likely due to overhead and allocation of threads when run on a single system. The fusion of both OPENMP and MPI would work best when data is scattered over several different devices over a closed network, or at least a system with multiple physical processors, since on one system/processor the overall load does not scale well and rather worsens the performance as compared to simple MPI implementation. Using more than one system would reduce this load when each device would implement OPENMP within its scope.

Project Recording:

<https://drive.google.com/file/d/1Agpkxp8vXYrNNFDd-9JxV9dVboz8KdB0/view?usp=share_link>